CONFIDENTIAL ASCII MSX

M S X -- D O S v e r s i o n 2

The advanced disk operating system

for MSX 2 computers

Copyright (1986) ASCII Corp.

CONTENTS

Page

1. Introduction 3
2. List of functions 4
3. Function By Function Definitions 7

3.1 \
. . \ Definitions of each function
. . /
3.90 /

This manual describes in detail the MSX-DOS function calls provided by MSX-DOS version 2.00.

Copyright (1986) ASCII Corporation
Copyright (1986) IS Systems Ltd.

1. INTRODUCTION

This document describes in detail each of the MSX-DOS 2.00 function calls. It should be read in conjunction with the "Program Interface specification" which describes system features such as file handles, fileinfo blocks and environment strings in general terms.

There are two ways of doing MSX-DOS function calls, reflecting the two different environments in which the system can run. Transient programs running in the MSX-DOS environment must access the functions with a "CALL 00005h" instruction. Disk BASIC and other MSX programs running in the disk BASIC environment (usually executing from ROM) must access the system via a "CALL 0F37Dh" instruction. It is NOT possible to mix these two types of call in one program since the environment will only be set up for one or the other.

There are some limitations when calling the system via 0F37Dh, particularly to do with error handling and abort routines. Also no parameters may be passed in page-1, unless they are in the master disk ROM (as they will be for disk BASIC) since the master disk ROM will be paged into page-1 when such a function call is made. The individual function descriptions mention the differences for particular functions.

3. LIST OF FUNCTIONS

Below there is a complete list of the functions calls. "CPM" indicates that the function is compatible with the equivalent CP/M 2.2 function, "MSX1" indicates compatibility with MSX-DOS version 1, and "NEW" indicates a function which is new to this system. An asterisk ("*") indicates that the function may be safely called from a user disk error routine (see function 64h and function 70h).

CPM MSX1		00h -	Program terminate
CPM MSX1	*	01h -	Console input
CPM MSX1	*	02h -	Console output
CPM MSX1	*	03h -	Auxiliary input
CPM MSX1	*	04h -	Auxiliary output
CPM MSX1	*	05h -	Printer output
CPM MSX1	*	06h -	Direct console I/O
MSX1	*	07h -	Direct console input
MSX1	*	08h -	Console input without echo
CPM MSX1	*	09h -	String output
CPM MSX1	*	0Ah -	Buffered line input
CPM MSX1	*	0Bh -	Console status
CPM MSX1	*	0Ch -	Return version number
CPM MSX1		0Dh -	Disk reset
CPM MSX1		0Eh -	Select disk
CPM MSX1		0Fh -	Open file (FCB)
CPM MSX1		10h -	Close file (FCB)
CPM MSX1		11h -	Search for first (FCB)
CPM MSX1		12h -	Search for next (FCB)
CPM MSX1		13h -	Delete file (FCB)
CPM MSX1		14h -	Sequential read (FCB)
CPM MSX1		15h -	Sequential write FCB)
CPM MSX1		16h -	Create file (FCB)
CPM MSX1		17h -	Rename file (FCB)
CPM MSX1	*	18h -	Get login vector
CPM MSX1	*	19h -	Get current drive
CPM MSX1		1Ah -	Set disk transfer address
MSX1		1Bh -	Get allocation information
		1Ch -	Unused
		1Dh -	Unused
		1Eh -	Unused
		1Fh -	Unused
		20h -	Unused

CPM MSX1		21h -	Random read (FCB)
CPM MSX1		22h -	Random write(FCB)
CPM MSX1		23h -	Get file size (FCB)
CPM MSX1		24h -	Set random record (FCB)
		25h -	Unused
MSX1		26h -	Random block write (FCB)
MSX1		27h -	Random block read (FCB)
CPM MSX1		28h -	Random write with zero fill (FCB)
		29h -	Unused
MSX1	*	2Ah -	Get date
MSX1	*	2Bh -	Set date
MSX1	*	2Ch -	Get time
MSX1	*	2Dh -	Set time
MSX1	*	2Eh -	Set/reset verify flag
MSX1		2Fh -	Absolute sector read
MSX1		30h -	Absolute sector write
NEW		31h -	Get disk parameters
		32h - \	
		. . \	Unused
		. . /	
		3Fh - /	
NEW		40h -	Find first entry
NEW		41h -	Find next entry
NEW		42h -	Find new entry
NEW		43h -	Open file handle
NEW		44h -	Create file handle
NEW		45h -	Close file handle
NEW		46h -	Ensure file handle
NEW		47h -	Duplicate file handle
NEW		48h -	Read from file handle
NEW		49h -	Write to file handle
NEW		4Ah -	Move file handle pointer
NEW		4Bh -	I/O control for devices
NEW		4Ch -	Test file handle
NEW		4Dh -	Delete file or subdirectory
NEW		4Eh -	Rename file or subdirectory
NEW		4Fh -	Move file or subdirectory
NEW		50h -	Get/set file attributes
NEW		51h -	Get/set file date and time
NEW		52h -	Delete file handle
NEW		53h -	Rename file handle
NEW		54h -	Move file handle
NEW		55h -	Get/set file handle attributes
NEW		56h -	Get/set file handle date and time

NEW	*	57h -	Get disk transfer address
NEW	*	58h -	Get verify flag setting
NEW		59h -	Get current directory
NEW		5Ah -	Change current directory
NEW		5Bh -	Parse pathname
NEW		5Ch -	Parse filename
NEW	*	5Dh -	Check character
NEW		5Eh -	Get whole path string
NEW		5Fh -	Flush disk buffers
NEW		60h -	Fork a child process
NEW		61h -	Rejoin parent process
NEW		62h -	Terminate with error code
NEW		63h -	Define abort exit routine
NEW		64h -	Define disk error handler routine
NEW	*	65h -	Get previous error code
NEW	*	66h -	Explain error code
NEW		67h -	Format a disk
NEW		68h -	Create or destroy RAM disk
NEW		69h -	Allocate sector buffers
NEW	*	6Ah -	Logical drive assignment
NEW	*	6Bh -	Get environment item
NEW	*	6Ch -	Set environment item
NEW	*	6Dh -	Find environment item
NEW	*	6Eh -	Get/set disk check status
NEW	*	6Fh -	Get MSX-DOS version number
NEW	*	70h -	Get/set redirection status

3. FUNCTION BY FUNCTION DEFINITIONS

Below are detailed descriptions of each of the MSX-DOS functions including both the old and new ones. The names in brackets after the function numbers are the public labels for the function codes which are defined in "CODES.MAC". Programs should use these names whenever possible.

Many of the functions below 40h return an error flag rather than an error code. If the error flag is set then the actual error code indicating the cause of the error can be obtained by the "get previous error code" function (function 65h). All of the functions above 40h return an error code in register A. The "Program Interface Specification" document describes the general errors which can be returned from many of the functions. The individual function specifications here describe the main error conditions which are specific to particular functions.

Note that many of the function calls which modify the information on a disk do not automatically flush disk buffers and so the disk is not necessarily correctly updated immediately after the function call is made. Such calls include all types of "create", "write", "delete", "rename", "change file attributes" and "change file date and time" function calls. The only functions which always flush disk buffers are "flush buffers", "close" and "ensure". After these operations the disk will always be correctly updated.

3.1 PROGRAM TERMINATE

```
Parameters:    C = 00h (_TERM0)
Results:       Does not return
```

This function terminates program with a zero return code. It is provided for compatibility, the preferred method of exiting a program is to use the "terminate with error code" function call (function 62h), passing a zero error code if that is what is desired. See the description of that function call, and also the "Program Interface Specification", for details of what happens when a program terminates. This function call never returns to the caller.

3.2 CONSOLE INPUT

```
Parameters:    C = 01h (_CONIN)
Results:     L=A = Character from keyboard
```

A character will be read from the standard input (file handle 0 - usually the keyboard) and echoed to the standard output (file handle 1 - usually the screen). If no character is ready then this function will wait for one. Various control characters, as specified for the "console status" function (function 0Bh), will be trapped by this function for various control purposes. If one of these characters is detected then it will be processed and this function will wait for another character. Thus these characters will never be returned to the user by this function.

3.3 CONSOLE OUTPUT

Parameters: C = 02h (_CONOUT)
E = Character to be output
Results: None

The character passed in register E is written to the standard output (file handle 1 - usually the screen). If printer echo is enabled then the character is also written to the printer. Various control codes and escape sequences are interpreted as screen control codes. A list of these is included in the "Program Interface Specification", they are a sub-set of the standard VT-52 control codes. TABs will be expanded to every eighth column.

A console input status check is done, and if any of the special control characters described for the "console status" function (function 0Bh) is found then it will be processed as for that function. Any other character will be saved internally for a later "console input" function call.

3.4 AUXILIARY INPUT

Parameters: C = 03h (_AUXIN)
Results: L=A = Input character

A character is read from the auxiliary input device (file handle 3) and if no character is ready then it will wait for one. The auxiliary input device must have been installed before this function may be used. If no such device has been installed then this function will always return the end of file character ("Ctrl-Z").

3.5 AUXILIARY OUTPUT

Parameters: C = 04h (_AUXOUT)
E = Character to be output

Results: None

The character passed in register E will be written to the auxiliary output device (file handle 3). The auxiliary output device must have been installed before this function may be used. If no such device has been installed then this function will simply throw the character away.

3.6 PRINTER OUTPUT

Parameters: C = 05h (_LSTOUT)
E = Character to be output

Results: None

The character passed in register E will be sent to the standard printer device (file handle 4 - usually the parallel printer). The same channel is used for console output which is echoed to the printer. TABs are not expanded by this function, although they are expanded when screen output is echoed to the printer with "Ctrl-P".

3.7 DIRECT CONSOLE I/O

Parameters: C = 06h (_DIRIO)
E = 00h...FEh - character for output
= FFh - requests input

Results: A=L = input - 00h - no character ready
else input character
undefined for output

If E=FFh on entry then the keyboard will be examined for a character and 00h returned if no character is ready. If a character is ready then it will be read from the standard input (file handle 0) and returned in register A without being echoed and with no check for control characters.

If E<>FFh on entry then the character in register E will be printed directly to the standard output (file handle 1) with no TAB expansion or printer echo. Also no console status check is done by this function. Note that although this function does not expand TABs, the VT-52 control codes include TAB expansion so the effect on the screen is the same. This may be different however if the standard screen driver is replaced.

3.8 DIRECT CONSOLE INPUT

Parameters: C = 07h (_DIRIN)
Results: L=A = Input character

This function is identical to the input option of function 06h, except that if no character is ready it will wait for one. Like function 06h, no echo or control characters checks will be done. This function is not compatible with CP/M which uses this function number for "get I/O byte".

3.9 CONSOLE INPUT WITHOUT ECHO

Parameters: C = 08h (_INNOE)
Results: L=A = Input character

This function is identical to function 01h except that the input character will not be echoed to the standard output. The same control character checks will be done. This function is not compatible with CP/M which uses this function number for "set I/O byte".

3.10 STRING OUTPUT

Parameters: C = 09h (_STROUT)
DE = Address of string
Results: None

The characters of the string pointed to by register DE will be output using the normal console output routine (function call 02h). The string is terminated by "$" (ASCII 24h).

3.11 BUFFERED LINE INPUT

Parameters: C = 0Ah (_BUFIN)
DE = Address of an input buffer
Results: None

DE must point to a buffer to be used for input. The first byte of this buffer must contain the number of characters which the buffer can hold (0...255). A line of input will be read from the standard input device (file handle 0 - usually the keyboard) and stored in the buffer. The input will be terminated when a CR is read from the standard input. The number of characters entered, which does not include the CR itself, will be stored at (DE+1). If there is room in the buffer then the CR will be stored after the last character.

When inputting from the keyboard (which will normally be the case), a simple line editor is provided, and also a 256 byte circular buffer of previous lines which can be edited and re-entered. The details of these editing facilities are described in the separate "Command Specification" document, so they are not included here. When the input buffer becomes full, the console bell will be rung for each character typed which cannot be put in the buffer. Each character entered will be echoed to the standard output and also to the printer if printer echo is enabled.

3.12 CONSOLE STATUS

```
Parameters:     C = 0Bh (_CONST)
Results:    L=A = 00h if no character ready
                = FFh if a character is ready
```

A flag is returned in register A to indicate whether a character is ready for input from the keyboard. If a character is ready then it will be read and tested for certain special control characters. If it is not one of these then it is stored in an internal single byte buffer and subsequent call to this function will return "character ready" immediately without checking the keyboard. If this function says that a character is ready then the character may be read by function 02h or 08h.

If the character is "Ctrl-C" then the program will be terminated with a ".CTRLC" error via the user's abort routine if one is defined. If the character is "Ctrl-P" then printer echo will be enabled and it will be disabled if it is "Ctrl-N". If the character is "Ctrl-S" then the routine will hang up waiting for another character to be pressed and then return "no character ready", thus providing a "wait" facility. The character typed to continue operation will be ignored, except that of it is "Ctrl-C" then the program will be terminated. These same input checks are also done for functions 01h, 02h, 08h, 09h and 0Ah.

3.13 RETURN VERSION NUMBER

```
Parameters:    C = 0Ch (_CPMVER)
Results:     L=A = 22h
             H=B = 00h
```

This function simply returns the CP/M version number which is being emulated. This is always version 2.2 in current systems.

3.14 DISK RESET

```
Parameters:    C = 0Dh (_DSKRST)
Results:       None
```

Any data which is waiting in internal buffers is written out to disk. It is not necessary to call this function in order to allow a disk change as is the case with CP/M. The disk transfer address is also set back to its default value of 80h by this function.

3.15 SELECT DISK

```
Parameters:    C = 0Eh (_SELDSK)
               E = Drive number.  0=A:  1=B:  etc.
Results:     L=A = Number of drives (1...26)
```

This function simply selects the specified drive as the default drive. The current drive is also stored at address 0004h for CP/M compatibility. The number of drives available is returned in register A and this will include the RAM disk if it is active.

3.16 OPEN FILE (FCB)

Parameters: C = 0Fh (_FOPEN)
DE = Pointer to unopened FCB

Results: L=A = 0FFh if file not found
= 0 if file is found

The unopened FCB must contain a drive which may be zero to indicate the default drive and a filename and extension which may be ambiguous. The current directory of the specified drive will be searched for a matching file and if found it will be opened. Matching entries which are sub-directories or system files will be ignored, and if the filename is ambiguous then the first suitable matching entry will be opened.

Device names may be put in the FCB (without a colon) to allow devices to be accessed as if they were actually disk files. The standard device names are defined in the "Program Interface Specification".

The low byte of the extent number is not altered by this function, and a file will only be opened if it is big enough to contain the specified extent. Normally the transient program will set the extent number to zero before calling this function. The high byte of the extent number will be set to zero to ensure compatibility with CP/M.

The filename and extension in the FCB will be replaced by the actual name of the file opened from the directory entry. This will normally be the same as what was there before but may be different if an ambiguous filename or one with lower case letters in was used.

The record count will be set to the number of 128 byte records in the specified extent, which is calculated from the file size. The file size field itself, the volume-id and the 8 reserved bytes will also be set up. The current record and random record fields will not be altered by this function, it is the program's responsibility to initialize them before using the read or write functions.

If the file cannot be found, then the "APPEND" environment item will be examined. If this is set then it is interpreted as a drive/path string which specifies a second directory in which to look for the file. The specified directory will be searched for the file and if found it will be opened as above. In this case the drive byte of the FCB will be set to the drive on which the file was found to ensure correct accessing of the file if the original drive byte was zero (default).

3.17 CLOSE FILE (FCB)

```
Parameters:    C = 10h (_FCLOSE)
              DE = Pointer to opened FCB
Results:     L=A = 0FFh if not successful
               =   0  if successful
```

The FCB must have previously been opened with either an OPEN or a CREATE function call. If the file has only been read then this function does nothing. If the file has been written to then any buffered data will be written to disk and the directory entry updated appropriately. The file may still be accessed after a close, so the function can be regarded as doing an "ensure" operation.

3.18 SEARCH FOR FIRST (FCB)

```
Parameters:    C = 11h (_SFIRST)
              DE = Pointer to unopened FCB
Results:     L=A = 0FFh if file not found
               =   0  if file found.
```

This function searches the current directory of the drive specified in the FCB (default drive if FCB contains zero) for a file which matches the filename and extension in the FCB. The filename may be ambiguous (containing "?" characters) in which case the first match will be found. The low byte of the extent field will be used, and a file will only be found if it is big enough to contain this extent number. Normally the extent field will be set to zero by the program before calling this function. System file and sub-directory entries will not be found.

If a suitable match is found (A=0) then the directory entry will be copied to the DTA address, preceded by the drive number. This can be used directly as an FCB for an OPEN function call if desired. The extent number will be set to the low byte of the extent from the search FCB, and the record count will be initialized appropriately (as for OPEN). The attributes byte from the directory entry will be stored in the S1 byte position, since its normal position (immediately after the filename extension field) is used for the extent byte.

If no match is found (A=0FFh) then the DTA will not be altered. In no case will the FCB pointed to by DE be modified at all. This function remembers sufficient information internally to allow it to continue the search with a SEARCH FOR NEXT function, so it is not necessary for the FCB to be preserved if doing a SEARCH FOR NEXT function.

In CP/M, if the drive number is set to "?" in this function then all directory entries, allocated or free will be matched. Also if the extent field is set to "?" then any extent of a file will be matched. Both of these features are normally only used by special purpose CP/M programs which are generally specific to the CP/M filing system (such as "STAT"). Neither feature is present in MSX-DOS 1.0 (although the documentation claims that they are). MSX-DOS 2 does not implement these features.

3.19 SEARCH FOR NEXT (FCB)

```
Parameters:     C = 12h (_SNEXT)
Results:      L=A = 0FFh if file not found
                  =   0  if file found.
```

This function must only be done after a SEARCH FOR FIRST has been done. It continues the search to look for the next match with the filename. The results returned from this function are identical to SEARCH FOR FIRST and all the same comments apply. The information used to continue the search is held internally within MSX-DOS and so the original FCB used in the SEARCH FOR FIRST need not still exist.

3.20 DELETE FILE (FCB)

```
Parameters:     C = 13h (_FDEL)
               DE = Pointer to unopened FCB
Results:      L=A = 0FFh if no files deleted
                  =   0  if files deleted OK
```

All files in the current directory of the disk specified by the FCB, and which match the ambiguous filename in the FCB, are deleted. Sub-directories, system files and read only files are not deleted. If any files at all are successfully deleted then this function returns with A=0. A return with A=FFh indicates that no files were deleted. Note that this function does not flush sector buffers to disk so the directory on disk may not have been fully updated when the function returns.

3.21 SEQUENTIAL READ (FCB)

```
Parameters:     C = 14h (_RDSEQ)
               DE = Pointer to opened FCB
Results:      L=A = 01h if error (end of file)
                  =   0  if read was successful
```

This function reads the next sequential 128 byte record from the file into the current disk transfer address. The record is defined by the current extent (high and low bytes) and the current record. After successfully reading the record, this function increments the current record and if it reaches 080h, sets it back to zero and increments the extent number. The record count field is also kept updated when necessary.

Unlike CP/M it is possible to have partially filled records, since the file size is not necessarily a multiple of 128 bytes. If this occurs then the partial record is padded out with zeroes when it is copied to the transient program's DTA address.

3.22 SEQUENTIAL WRITE (FCB)

```
Parameters:    C = 15h (_WRSEQ)
              DE = Pointer to opened FCB
Results:    L=A = 01h if error (disk full)
              = 0  if write was successful
```

This function writes the 128 bytes from the current disk transfer address to the file at the position defined by the current record and extent, which are then incremented appropriately. The record count byte is kept updated correctly if the file is extended or if the write moves into a new extent. The file size in the FCB is also updated if the file is extended.

3.23 CREATE FILE (FCB)

```
Parameters:    C = 16h (_FMAKE)
              DE = Pointer to unopened FCB
Results:    L=A = 0FFh if unsuccessful
              = 0  if successful
```

This function creates a new file in the current directory of the specified drive and opens it ready for reading and writing. The drive, filename and low byte of the extent number must be set up in the FCB and the filename must not be ambiguous. Checks will be done to ensure that invalid filenames are not created.

If there is already a file of the required name then the action depends on the value of the extent number byte. Normally this will be zero and in this case the old file will be deleted and a new one created. However if the extent number is non-zero then the existing file will be opened without creating a new file. This ensures compatibility with early versions of CP/M where each extent had to be explicitly created.

In all cases the resulting file will be opened with the required extent number exactly as if an OPEN function call had been done.

3.24 RENAME FILE (FCB)

```
Parameters:     C = 17h (_FREN)
               DE = Pointer to modified unopened FCB
Results:     L=A = 0FFh not if successful
                 =   0  if successful
```

The modified unopened FCB has the normal drive and filename, and also a second filename starting at (DE+17). Every file in the current directory of the specified drive which matches the first filename, is changed to the second filename with "?" characters in the second filename leaving the appropriate character unchanged. Checks are done to prevent duplicate or illegal filenames from being created. Entries for sub-directories and system files will not be renamed.

Note that this function does not flush sector buffers to disk so the directory on disk may not have been fully updated when the function returns.

3.25 GET LOGIN VECTOR

```
Parameters:     C = 18h (_LOGIN)
Results:       HL = Login vector
```

This function returns a bit set in HL for each drive which is available, bit-0 of L corresponding to drive "A:". Up to eight drives ("A:" to "H:") are supported by the system currently, so register H will usually be zero on return.

3.26 GET CURRENT DRIVE

```
Parameters:     C = 19h (_CURDRV)
Results:      L=A = Current default drive (0=A: etc)
```

This function just returns the current drive number.

3.27 SET DISK TRANSFER ADDRESS

```
Parameters:     C = 1Ah (_SETDTA)
               DE = Required Disk Transfer Address

Results:        None
```

This function simply records the address passed in DE as the disk transfer address. This address will be used for all subsequent FCB read and write calls, for "search for first" and "search for next" calls to store the directory entry, and for absolute read and write calls. It is not used by the new MSX-DOS read and write functions. The address is set back to 80h by a DISK RESET call.

3.28 GET ALLOCATION INFORMATION

```
Parameters:     C = 1Bh (_ALLOC)
                E = Drive number (0=default, 1=A: etc)

Results:        A = Sectors per cluster
               BC = Sector size (always 512)
               DE = Total clusters on disk
               HL = Free clusters on disk
               IX = Pointer to DPB
               IY = Pointer to first FAT sector
```

This function returns various information about the disk in the specified drive. It is not compatible with CP/M which uses this function number to return the address of an allocation vector. Note that unlike MSX-DOS version 1, only the first sector of the FAT may be accessed from the address in IY, and the data there will only remain valid until the next MSX-DOS call.

3.29 RANDOM READ (FCB)

```
Parameters:    C = 21h (_RDRND)
              DE = Pointer to opened FCB
Results:     L=A = 01h if error (end of file)
                 0  if read was successful
```

This function reads a 128 byte record from the file to the current disk transfer address. The file position is defined by the three byte random record number in the FCB (bytes 21h...23h). Unlike CP/M all three bytes of the random record number are used. A partial record at the end of the file will be padded with zeroes before being copied to the user's DTA.

The random record number is not altered so successive calls to to this function will read the same record unless the transient program alters the random record number. A side effect is that the current record and extent are set up to refer to the same record as the random record number. This means that sequential reads (or writes) can follow a random read and will start from the same record. The record count byte is also set up correctly for the extent.

3.30 RANDOM WRITE (FCB)

```
Parameters:    C = 22h (_WRRND)
              DE = Pointer to opened FCB
Results:     L=A = 01h if error (disk full)
                 =  0  if no error
```

This function writes a 128 byte record from the current disk transfer address to the file, at the record position specified by the three byte random record number (bytes 21h...23h). All three bytes of the random record number are used. If the record position is beyond the current end of file then un-initialized disk space will be allocated to fill the gap.

The random record number field will not be changed, but the current record and extent fields will be set up to refer to the same record. The record count byte will be adjusted as necessary if the file is being extended or if the write goes into a new extent.

3.31 GET FILE SIZE (FCB)

```
Parameters:    C = 23h (_FSIZE)
              DE = Pointer to unopened FCB
Results:     L=A = 0FFh if file not found
                 =   0  if file found OK
```

This function searches for the first match with the filename in the FCB, exactly the same as OPEN FILE (function 0Fh). The size of the located file is rounded up to the nearest 128 bytes and the number of records determined. The three byte random record field of the FCB is set to the number of records, so it is the number of the first record which does not exist. The fourth byte of the random record number is not altered.

3.32 SET RANDOM RECORD (FCB)

Parameters: C = 24h (_SETRND)
DE = Pointer to opened FCB
Results: None

This function simply sets the three byte random record field in the FCB to the record determined by the current record and extent number. The fourth byte of the random record number is not altered. No check is done as to whether the record actually exists in the file.

3.33 RANDOM BLOCK WRITE (FCB)

Parameters: C = 26h (_WRBLK)
DE = Pointer to opened FCB
HL = Number of records to write
Results: A = 01h if error
= 0 if no error

Data is written from the current disk transfer address to the position in the file defined by the random record number. The record size is determined by the record size field in the FCB (bytes 0Eh and 0Fh) which must be set by the user after opening the file and before calling this function. If the record size is less than 64 bytes then all four bytes of the random record number are used, otherwise only the first three are used.

The number of records to be written is specified by HL, and together with the record size this determines the amount of data to be written. An error will be returned if the write attempts to go above the top of memory (0FFFFh), thus limiting the maximum size of a transfer.

After writing the data, the random record field is adjusted to the next record number in the file (ie. HL is added on to it). The current record and extent fields are not used or altered. The file size field is updated if the file has been extended.

The record size can be any value from 1...0FFFFh. Small record sizes are no less efficient that large record sizes so if desired the record size can be set to one and the record count then becomes a byte count. It is desirable to read as much as possible with one function call since one large transfer will be quicker than several small ones.

If the number of records to write (HL) is zero then no data will be written, but the size of the file will be altered to the value specified by the random record field. This may be either longer or shorter than the file's current size and disk space will be allocated or freed as required. Additional disk space allocated in this way will not be initialized to any particular value.

3.34 RANDOM BLOCK READ (FCB)

Parameters:	C = 27h (_RDBLK)
	DE = Pointer to opened FCB
	HL = Number of records to read
Results:	A = 01h if error
	= 0 if no error
	HL = Number of records actually read

This function is the complement of the BLOCK WRITE function described above and most of the same comments apply as regards its use. Again if large blocks are read then it will be much faster than the normal CP/M operation.

For example if it is desired to read 20k from a file, it is better to read the 20k with one function call rather than 20 separate function calls of 1k each. However it makes no difference whether the 20k read is done with a record size of 1 and a record count of 20k, with a record size of 20k and a record count of 1, or any intermediate combination.

The number of records actually read is returned in HL. This may be smaller than the number of records requested if the end of the file was encountered. In this case any partial record will be padded out with zeroes before being copied to the users DTA. The random record field is adjusted to the first record not read, ie. the value returned in HL is added on to it.

3.35 RANDOM WRITE WITH ZERO FILL (FCB)

Parameters:	C	= 28h (_WRZER)
	DE	= Pointer to opened FCB
Results:	L=A	= 01h if error
		= 00h if no error

This function is identical to RANDOM WRITE (function 22h) except that if the file has to be extended, any extra allocated disk clusters will be filled with zeroes before writing the data.

3.36 GET DATE

```
Parameters:   C = 2Ah (_GDATE)
Results:     HL = Year 1980...2079
              D = Month (1=Jan...12=Dec)
              E = Date (1...31)
              A = Day of week (0=Sun...6=Sat)
```

This function simply returns the current value of the internal calender in the format shown.

3.37 SET DATE

```
Parameters:   C = 2Bh (_SDATE)
             HL = Year 1980...2079
              D = Month (1=Jan...12=Dec)
              E = Date (1...31)
Results:      A = 00h if date was valid
                  FFh if date was invalid
```

The supplied date is checked for validity and if it is valid then it is stored as the new date. The validity checks include full checking for the number of days in each month and leap years. If the date is invalid then the current date will be left unaltered. The date is stored in the real time clock chip so it will be remembered when the machine is turned off.

3.38 GET TIME

```
Parameters:   C = 2Ch (_GTIME)
Results:      H = Hours (0...23)
              L = Minutes (0...59)
              D = Seconds (0...59)
              E = Centiseconds (always zero)
```

This function returns the current value of the system clock in the format shown.

3.39 SET TIME

```
Parameters:   C = 2Dh (_STIME)
              H = Hours (0...23)
              L = Minutes (0...59)
              D = Seconds (0...59)
              E = Centiseconds (ignored)
Results:      A = 00h if time was valid
                  FFh if time was invalid
```

This function sets the internal system clock to the specified time value. If the time is invalid then register A will be returned as 0FFh to indicate an error and the current time will be left unaltered. The time is stored in the real time clock chip and so it will be remembered and kept correct when the machine is turned off.

3.40 SET/RESET VERIFY FLAG

Parameters:	C = 2Eh (_VERIFY) E = 0 to disable verify <> 0 to enable verify
Results:	None

This function simply enables or disables automatic verification of all writes. It defaults to off when MSX-DOS is started up. Enabling verify improves system reliability slightly but also slows down write operations considerably.

3.41 ABSOLUTE SECTOR READ

```
Parameters:   C = 2Fh (_RDABS)
             DE = Sector number
              L = Drive number (0=A: etc.)
              H = Number of sectors to read
Results:      A = Error code (0=> no error)
```

This function reads sectors directly from the disk without interpreting them as files. The disk must be a valid DOS disk in order for the sector number to be translated into a physical position on the disk. The sectors will be read to the current disk transfer address. Any disk error will be reported by the system in the usual way.

3.42 ABSOLUTE SECTOR WRITE

```
Parameters:   C = 30h (_WRABS)
             DE = Sector number
              L = Drive number (0=A: etc.)
              H = Number of sectors to write
Results:      A = Error code
```

This function writes sectors directly to the disk without interpreting them as files. The disk must be a valid DOS disk in order for the sector number to be translated into a physical position on the disk. The sectors will be written from the current disk transfer address. Any disk errors are reported by the system in the usual way.

3.43 GET DISK PARAMETERS

```
Parameters:   C = 31h (_DPARM)
             DE = Pointer to 32 byte buffer
              L = Drive number (0=default, 1=A: etc.)
Results:      A = Error code
             DE = Preserved
```

This functions returns a series of parameters relating to the format of the disk in the specified drive. It is useful for programs which are going to do absolute sector reads and writes, in order for them to be able to interpret the absolute sector numbers. The parameters returned contain some redundant information in order to provide parameters which are most useful to transient programs. The format of the returned parameter block is:

```
DE+0       - Physical drive number (1=A: etc)
DE+1,2     - Sector size (always 512 currently)
DE+3       - Sectors per cluster (non-zero power of 2)
DE+4,5     - Number of reserved sectors (usually 1)
DE+6       - Number of copies of the FAT (usually 2)
DE+7,8     - Number of root directory entries
DE+9,10    - Total number of logical sectors
DE+11      - Media descriptor byte
DE+12      - Number of sectors per FAT
DE+13..14  - First root directory sector number
DE+15..16  - First data sector number
DE+17..18  - Maximum cluster number
DE+19      - Dirty disk flag
DE+20..23  - Volume id. (-1 => no volume id.)
DE+24..31  - Reserved (currently always zero)
```

3.44 FIND FIRST ENTRY

Parameters:
C = 40h (_FFIRST)
DE = Drive/path/file ASCIIZ string or fileinfo block pointer
HL = filename ASCIIZ string (only if DE = fileinfo pointer)
B = Search attributes
IX = Pointer to new fileinfo block

Results:
A = Error
(IX) = Filled in with matching entry

The "drive/path" portion of the string, or the fileinfo block, specifies a directory which is to be searched. A ".IATTR" error will be returned if a fileinfo block which specifies a file is passed. The "file" portion of the string, or the filename string in HL, determines what filenames will be matched. If no match is found then a ".NOFIL" error is returned, otherwise the fileinfo block pointed to by IX is filled in with the details of the matching entry.

The filename may contain ambiguous filename characters ("?" or "*") in which case the first matching entry will be returned. If the filename is null (either the ASCIIZ string pointed to by DE is null or ends in a "\" or the filename string pointed to by HL is null), then this function will behave exactly as if the filename was "*.*" so any name will match.

The attributes byte in register B specifies what type of entry will be matched. If it is zero then only non-hidden, non-system files will be found. If the directory, hidden or system bits in register B are set then entries with these attributes will be matched as well as ordinary files. The read only and archive bits of register B are ignored.

If the volume name bit of register B is set then the search is exclusive, only the volume label entry will be found. In this case also the fileinfo block and filename or the drive/path/file string are ignored apart from specifying the drive. This means that the volume name will always be found in the root directory if it exists whether or not it matches the filename given.

If DE points to a fileinfo block, then if desired, IX can point to the same fileinfo block. In this case when a match is found the new fileinfo block will overwrite the old one.

3.45 FIND NEXT ENTRY

```
Parameters:     C = 41h (_FNEXT)
               IX = Pointer to fileinfo block from
                    previous find first function.
Results:        A = Error
             (IX) = Filled in with next matching entry
```

This function should only be used after a "find first" function call. It searches the directory for the next match to the (presumably ambiguous) filename which was given to the "find first" function call. If there are no more matching entries then a ".NOFIL" error is returned, otherwise the fileinfo block is filled in with the information about the new matching entry.

3.46 FIND NEW ENTRY

```
Parameters:     C = 42h (_FNEW)
               DE = Drive/path/file ASCIIZ string
                    or fileinfo block pointer
               HL = filename ASCIIZ string (only if
                    DE = fileinfo pointer)
                B = b0..b6 = Required attributes
                    b7 = Create new flag
               IX = Pointer to new fileinfo block
                    containing template filename
Results:        A = Error
             (IX) = Filled in with new entry
```

This function is very similar to the "find first" function described above. The parameters in HL and DE are used in exactly the same way to specify a directory entry. However instead of searching the selected directory for an entry which matches the specified name, a new entry will be created with this name. The fileinfo block pointed to by IX will be filled in with information about the new entry just as if it had been found with a "find first" call.

If there are any ambiguous characters ("?" or "*") in the filename, then they will be replaced by the appropriate character from a "template filename" in the filename position of the new fileinfo block pointed to by IX. If the result is still ambiguous, or otherwise illegal, then a ".IFNM" error is returned. This is useful for copy operations which do an automatic rename.

Like "find first", if the filename is null, then it will be treated exactly as if it was "*.*". For this function that means that the template filename will be used as the new filename to create.

A ".DRFUL" error will be returned if there is no room in the root directory, and a ".DKFUL" if a sub-directory must be extended and the disk is already full.

The attribute byte passed in register B is the attribute which the new entry will be given. If the volume name bit is set then a volume name will be created in the root directory. If the directory bit is set then the entry created will be for a sub-directory, otherwise it will be for a file. The system, hidden and read only bits may be set for a file, and the hidden bit for a sub-directory. A file will always be created with the archive attribute bit set.

A file will be created as zero length with the current date and time. A sub-directory will have a single cluster allocated to it and the "." and ".." entries will be initialized appropriately. These two entries will be marked as hidden directories.

If there is already an entry with the required name in the directory then the action depends on the "create new" flag (bit-7 of register B) and also on the type of the entry. If the "create new" flag is set then a ".FILEX" error will always be returned. Setting this flag therefore ensures that an existing file will not be deleted.

If an entry already exists and the "create new" flag is not set then the type of the existing entry is examined to see whether it can be deleted to make room for the new file. An error will be returned if the entry is a read only file (".FILRO" error), a system file (".SYSX" error) or a sub-directory (".DIRX" error) or there is a file handle already open to this file (".FOPEN" error). If we are trying to create a sub-directory then even an ordinary file will not be deleted (".FILEX" error).

For all of these error codes (".FILEX", ".FILRO", ".SYSX", ".DIRX", ".FOPEN"), the fileinfo block will be filed in with the details of the already existing entry and this fileinfo block may be used exactly as if it had been returned from a "find first" function.

3.47 OPEN FILE HANDLE

```
Parameters:   C = 43h (_OPEN)
             DE = Drive/path/file ASCIIZ string
                  or fileinfo block pointer
              A = Open mode. b0 set => no write
                             b1 set => no read
                             b2 set => inheritable
                         b3..b7   -  must be clear

Results:      A = Error
              B = New file handle
```

The drive/path/file string or the fileinfo block should normally refer to a file rather than a sub-directory or volume name. If it is a volume name then a ".IATTR" error will be returned. If it is a sub-directory then no error will be returned, but register B will be 0FFh which is never a valid file handle.

Assuming that a file is specified then it will be opened ready for reading and/or writing (depending on the open mode in A) and a new file handle for it will be returned in register B. The lowest available file handle number will be used and an error will result if there are no spare file handles (".NHAND" error), or insufficient memory (".NORAM" error).

If the "no read" bit of register A is set then reads from the file handle will be rejected and if the "no write" bit is set then writes will be rejected, in both cases with an ".ACCV" error. Writes will also be rejected if the file is read only (".FILRO" error). If the "inheritable" bit of register A is set then the file handle will be inherited by a new process created by the "fork" function call (see function 60h).

If a device file handle is opened by giving a filename which matches one of the built in devices (for example "CON" or "NUL"), then it will always be opened initially in ASCII mode. The IOCTL function (function 4Bh) can be used to change this to binary mode but great care must be taken in reading from devices in binary mode because there is no end of file condition.

Copyright (1986) ASCII Corp.

3.48 CREATE FILE HANDLE

```
Parameters:   C = 44h (_CREATE)
             DE = Drive/path/file ASCIIZ string
              A = Open mode. b0 set => no write
                             b1 set => no read
                             b2 set => inheritable
                          b3..b7    - must be clear
              B = b0..b6 = Required attributes
                   b7 = Create new flag
Results:      A = Error
              B = New file handle
```

A file or sub-directory, as specified by the attributes in register B, will be created with the name and in the directory specified by the drive/path/file string. A ".IATTR" error is returned if register B specifies a volume name.

An error will be returned if the file or sub-directory cannot be created. The error conditions in this case are the same as for the "find new" function (function 42h) with the main error codes being ".FILEX", ".DIRX", ".SYSX", ".FILRO", ".FOPEN", ".DRFUL" and ".DKFUL". Like the "find new" function, if the "create new" flag (bit-7 of register B) is set then an existing file will not be deleted and will always return a ".FILEX" error.

If the attributes byte specifies a sub-directory then the hidden bit may also be set to create a hidden sub-directory. For a file, the hidden, system or read only bits may be set to create a file with the appropriate attributes. An invalid attributes bits will simply be ignored. A file will always be created with the archive attribute bit set.

A file will automatically be opened just as for the "open" function described above, and a file handle returned in register B. The "open mode" parameter is interpreted in the same way as for the "open" function. A sub-directory will not be opened (because this is meaningless) so register B will be returned as 0FFh which can never be a valid file handle.

3.49 CLOSE FILE HANDLE

```
Parameters:   C = 45h (_CLOSE)
              B = File handle
Results:      A = Error
```

This function releases the specified file handle for re-use. If the associated file has been written to then its directory entry will be updated with a new date and time, the archive attributes bit will be set, and any buffered data will be flushed to disk. Any subsequent attempt to use this file handle will return an error. If there are any other copies of this file handle, created by "duplicate file handle" or "fork", then these other copies may still be used.

3.50 ENSURE FILE HANDLE

Parameters:	C = 46h (_ENSURE)
	B = File handle
Results:	A = Error

If the file associated with the file handle has been written to then its directory entry will be updated with a new date and time, the archive attributes bit will be set, and any buffered data will be flushed to disk. The file handle is not released and so it can still be used for accessing the file, and the current file pointer setting will not be altered.

3.51 DUPLICATE FILE HANDLE

Parameters:	C = 47h (_DUP)
	B = File handle
Results:	A = Error
	B = New file handle

This function creates a copy of the specified file handle. The lowest available file handle number will always be used and a ".NHAND" error returned if there are none available. The new file handle will refer to the same file as the original and either one may be used. If the file pointer of one handle is moved, the other one will also be moved. If either handle is closed the other one may still be used.

Note that because duplicate file handles created by this function are not "separately opened", they do not count as separate file handles for the purposes of generating ".FOPEN" errors. So for example a "DUP"ed file handle may be renamed (function 53h) or have its attributes changed (function 55h) and the effect will apply to both file handles. Note in particular that if one copy of a "DUP"ed file handle is deleted (function 54h) then the file really will be deleted and the other file handle, although still open, can no longer be used safely. If it is used (other than being closed, ensured or deleted) then an ".HDEAD" error will be returned.

3.52 READ FROM FILE HANDLE

Parameters:	C = 48h (_READ)
	B = File handle
	DE = Buffer address
	HL = Number of bytes to read
Results:	A = Error
	HL = Number of bytes actually read

The specified number of bytes are read from the file at the current file pointer position and copied to the buffer address specified in register DE. The file pointer is then updated to the next sequential byte. A ".ACCV" error will be returned if the file handle was opened with the "no read" access bit set.

The number of bytes read may be less than the number requested for various reasons, and the number read will be returned in register HL if there is no error. In general if less is read than requested then this should not be treated as an error condition but another read should be done to read the next portion, until a ".EOF" error is returned. An ".EOF" error will never be returned for a partial read, only for a read which reads zero bytes. Reading files in this way ensures that device file handles will work correctly (see below).

For disk files the number of bytes read will only be less than the number requested if the end of the file is reached and in this case the next read operation will read zero bytes and will return an ".EOF" error. When reading from a device file handle (for example the standard file handles 0 to 4), the behaviour depends on the particular device, and on whether it is being read in ASCII or binary mode (see function 4Bh below). The "CON" device will be described as an example because it is the most commonly used device, but other devices behave similarly.

When reading from the "CON" device in binary mode, characters will be read from the keyboard, without any interpretation and without being echoed to the screen or printer. The exact number of characters requested will always be read and there is no end of file condition. Because of the lack of any end of file indication, great care must be taken when reading from devices in binary mode.

A read function call to the "CON" device in ASCII mode (the default mode and that which normally applies to the standard input channel), will only read one line of input. The input line will be read from the keyboard with the normal line editing facilities available to the user, and the character typed will be echoed to the screen and to the printer if Ctrl-P is enabled. Special control characters "Ctrl-P", "Ctrl-N", "Ctrl-S" and "Ctrl-C" will be tested for and will be treated exactly as for function 0Bh.

When the user types a carriage return the line will be copied to the read buffer, terminated with a CR-LF sequence and the read function will return with an appropriate byte count. The next read will start another buffered line input operation. If the number of bytes requested in the read was less than the length of the line input then as many character as requested will be returned, and the next read function call will return immediately with the next portion of the line until it has all been read.

If the user types a line which starts with a "Ctrl-Z" character then this will be interpreted as indicating end of file. This single "Ctrl-Z" will be returned as the only character read and the rest of the line will be discarded including the CR-LF sequence. The next read function call on this file handle will read zero bytes and return an ".EOF" error. A subsequent read after this will be back to normal and will start another line input. The end of file condition is thus not permanent.

3.53 WRITE TO FILE HANDLE

Parameters:	C = 49h (_WRITE)
	B = File handle
	DE = Buffer address
	HL = Number of bytes to write
Results:	A = Error
	HL = Number of bytes actually written

This function is very similar to the "read" function above (function 48h). The number of bytes specified will be written to the current file pointer position in the file, and the file pointer will be adjusted to point to just after the last byte written. If the file was opened with the "no write" access bit set then a ".ACCV" error will be returned, and if the file is read only then a ".FILRO" error will be returned.

If the write goes beyond the current end of file then the file will be extended as necessary. If the file pointer is already beyond the end of the file then disk space will be allocated to fill the gap and will not be initialized. If there is insufficient disk space then a ".DKFUL" error will be returned and no data will be written, even if there was room for some of the data.

The number of bytes written can usually be ignored since it will either be zero if an error is returned or it will be equal to the number requested if the write was successful. It is very much more efficient to write files in a few large blocks rather than many small ones, so programs should always try to write in as large blocks as possible.

This function sets a "modified" bit for the file handle which ensures that when the file handle is closed or ensured, either explicitly or implicitly, the directory entry will be updated with the new date, time and allocation information. Also the archive bit will be set to indicate that this file has been modified since it was last archived.

Writing to device file handles is not a complicated as reading from them because there are no end of file conditions or line input to worry about. There are some differences between ASCII and binary mode when writing to the "CON" device, in that a console status check is done in ASCII mode only. Also printer echo if enabled will only be done in ASCII mode.

3.54 MOVE FILE HANDLE POINTER

Parameters:
C = 4Ah (_SEEK)
B = File handle
A = Method code
DE:HL = Signed offset

Results:
A = Error
DE:HL = New file pointer

The file pointer associated with the specified file handle will be altered according to the method code and offset, and the new pointer value returned in DE:HL. The method code specifies where the signed offset is relative to as follows:

A=0 Relative to the beginning of the file
A=1 Relative to the current position
A=2 Relative to the end of the file.

Note that an offset of zero with an method code of 1 will simply return the current pointer value, and with a method code of 2 will return the size of the file. No end of file check is done so it is quite possible (and sometimes useful) to set the file pointer beyond the end of the file. If there are any copies of this file handle created by the "duplicate file handle" function (function 47h) or the "fork" function (function 60h) then their file pointer will also be changed.

The file pointer only has any real meaning on disk files since random access is possible. On device files the file pointer is updated appropriately when any read or write is done, and can be examined or altered by this function. However changing will have no effect and examining it is very unlikely to be useful.

3.55 I/O CONTROL FOR DEVICES

Parameters:
C = 4Bh (_IOCTL)
B = File handle
A = Sub-function code
00h => get file handle status
01h => set ASCII/binary mode
02h => test input ready
03h => test output ready
04h => find screen size
DE = Other parameters

Results:
A = Error
DE = Other results

This function allows various aspects of file handles to be examined and altered. In particular it can be used to determine whether a file handle refers to a disk file or a device. This is useful for programs which want to behave differently for disk files and device I/O.

This function is passed the file handle in register B and a sub-function code in register A which specifies one of various different operations. Any other parameters required by the particular sub-function are passed in register DE and results are returned in register DE. If the sub-function code is invalid then a ".ISBFN" error will be returned.

If A=0 then the operation is "get file handle status". This returns a word of flags which give various information about the file handle. The format of this word is different for device file handles and disk file handles, and bit-7 specifies which it is. The format of the word is as follows:

```
For devices:     DE -      b0  set  => console input device
                           b1  set  => console output device
                       b2..b4  reserved
                           b5  set  => ASCII mode
                               clear=> binary mode
                           b6  set  => end of file
                           b7  always set (=> device)
                      b8..b15  reserved

For disk files: DE - b0..b5  drive number (0=A: etc)
                           b6  set  => end of file
                           b7  always clear (=> disk file)
                      b8..b15  reserved
```

Note that the end of file flag is the same for devices as for disk files. For devices it will be set if the next attempt to read from the device would produce a ".EOF" error and will be cleared by that read. Thus immediately after getting a ".EOF" error from a device file handle, this end of file flag will already have been cleared. For disk files it is worked out on each call to this function by comparing the file pointer with the file size.

If A=1 then the operation is a "set ASCII/binary mode". This operation is only allowed for device file handles. An ASCII/binary flag must be passed in bit-5 of register E (exactly where it is returned by "get file handle status". This is set for ASCII mode and clear for binary mode. All other bits of register DE are ignored.

If A=2 or 3 then the operation is "test input ready" or "test output ready" respectively. In both cases a flag is returned in register E which is FFh if the file handle is ready for a character and 00h if not. The exact meaning of "ready for a character" depends on the device. Disk file handles are always ready for output, and are always ready for input unless the file pointer is at the end of file. The "CON" device checks the keyboard status to determine whether it is ready for input or not.

If A=4 the the operation is "get screen size". This returns the logical screen size for the file handle with the number of rows in register D and the number of columns in register E. For devices with no screen size (such as disk files) both D and E will be zero. Zero for either result should therefore be interpreted as "unlimited". For example this function is used by the "DIR /W" command to decide how many files to print per line, and a value of zero for register E is defaulted to 80.

3.56 TEST FILE HANDLE

```
Parameters:     C = 4Ch (_HTEST)
                B = File handle
               DE = Drive/path/file ASCIIZ string
                       or fileinfo block pointer
Results:        A = Error
                B = 00h => not the same file
                    FFh => same file
```

This rather specialist function is passed a file handle and either a drive/path/file string or a fileinfo block which identifies a file. It determines if the two files are actually the same file and returns a flag indicating the result. Note that if the file handle is for a device rather than a disk file then it will always return "B=00h" to indicate "not the same file".

This function allows the "COPY" command to detect certain error conditions such as copying file onto themselves and give the user informative error messages. It may also be useful for other programs which need to do similar tests.

3.57 DELETE FILE OR SUBDIRECTORY

Parameters: C = 4Dh (_DELETE)
DE = Drive/path/file ASCIIZ string
or fileinfo block pointer
Results: A = Error

This function deletes the object (file or sub-directory) specified by the drive/path/file string or the fileinfo block. Global filename characters are not allowed so only one file or sub-directory can be deleted with this function. A sub-directory can only be deleted if it is empty (".DIRNE") error if not). The "." and ".." entries in a sub-directory cannot be deleted (".DOT" error) and neither can the root directory. A file cannot be deleted if there is a file handle open to it (.FOPEN error) or if it is read only (.FILRO error).

If it is a file then any disk space which was allocated to it will be freed. If the disk is an MSX-DOS 2.00 disk then enough information is retained on the disk to allow the "UNDEL" utility program do undelete the file. This information is only retained until the next disk space allocation (usually a write to a file) is done on this disk. After making this function call, if a fileinfo block was passed then it must not be used again (other than passing it to a "find "next" function) since the file to which it refers no longer exists.

If a device name such as "CON" is specified then no error will be returned but the device will not actually be deleted.

3.58 RENAME FILE OR SUBDIRECTORY

Parameters: C = 4Dh (_RENAME)
DE = Drive/path/file ASCIIZ string
or fileinfo block pointer
HL = New filename ASCIIZ string
Results: A = Error

This function renames the object (file or sub-directory) specified by the drive/path/file string or the fileinfo block, with the new name in the string pointed to by HL. The new filename string must not contain a drive letter or directory path (".IFNM" error if it does). If a device name such as "CON" is specified then no error will be returned but the device will not actually be renamed.

Global filename characters are not allowed in the drive/path/file string, so only one object can be renamed by this function. However global filename characters are allowed in the new filename passed in HL and where they occur the existing filename character will be left unaltered. Checks are done to avoid creating an illegal filename, for example a file called "XYZ" cannot be renamed with a new filename string of "????A" because the new filename would be "XYZ A" which is illegal. In this case a ".IFNM" error will be returned.

If there is already an entry with the new filename then an error (".DUPF") is returned to avoid creating duplicate filenames. The "." and ".." entries in a sub-directory cannot be renamed (".IDOT" error) and neither can the root directory (it has no name). A file cannot be renamed if there is a file handle open to it (".FOPEN" error) although a read only file can be renamed.

Note that if DE pointed to a fileinfo block, this is not updated with the new name of the file. Therefore care must be taken in using the fileinfo block after making this function call.

3.59 MOVE FILE OR SUBDIRECTORY

Parameters:	C = 4Fh (_MOVE)
	DE = Drive/path/file ASCIIZ string or fileinfo block pointer
	HL = New path ASCIIZ string
Results:	A = Error

This function moves the object (file or sub-directory) specified by the drive/path/file string or the fileinfo block, to the directory specified by the new path string pointed to by HL. There must not be a drive name in the new path string. If a device name such as "CON" is specified then no error will be returned but the device will not actually be moved.

Global filename characters are not allowed in any of the strings so only one object can be moved by this function, although if a sub-directory is moved, all its descendants will be moved with it. If there is already an entry of the required name in the target directory then a ".DUPF" error is returned to prevent creating duplicate filenames. The "." and ".." entries in a sub-directory cannot be moved (".DOT" error) and also a directory cannot be moved into one of its own descendants (".DIRE" error) since this would create an isolated loop in the filing system. A file cannot be moved if there is a file handle open to it (".FOPEN" error).

Note that if a fileinfo block is passed to this function, the internal information in the fileinfo block is not updated to reflect the new location of the file. This is necessary because otherwise the fileinfo block could not be used for a subsequent "find next" function call. However it does mean that the fileinfo block no longer refers to the moved file and so must not be used for any operations on it such as "rename" or "open".

3.60 GET/SET FILE ATTRIBUTES

```
Parameters:   C = 50h (_ATTR)
              DE = Drive/path/file ASCIIZ string
                   or fileinfo block pointer
              A = 0 => get attributes
                  1 => set attributes
              L = New attributes byte (only if A=1)
Results:      A = Error
              L = Current attributes byte
```

This function is normally used to change the attributes of a file or sub-directory. It can also be used to find out the current attributes but this is more usually done with the "find first" function (function 40h). If A=0 then the current attributes byte for the file or sub-directory will just be returned in register L.

If A=1 then the attributes byte will be set to the new value specified in register L, and this new value will also be returned in register L. Only the system, hidden, read only and archive bits may be altered for a file, and only the hidden bit for a sub-directory. An ".IATTR" error will be returned if an attempt is made to alter any other attribute bits. If a fileinfo block is passed then the attributes byte in it will not be updated with the new setting.

Global filename characters are not allowed so only one object can have its attributes set by this function. The attributes of the root directory cannot be changed because it does not have any. The attributes of a file cannot be changed if there is a file handle open to it (".FOPEN" error). The attributes of the "." and ".." directory entries however can be changed. If a device name such as "CON" is specified then no error will be returned but the device's attributes will not actually be changed (since it does not have any).

3.61 GET/SET FILE DATE AND TIME

Parameters:
C = 51h (_FTIME)
DE = Drive/path/file ASCIIZ string
or fileinfo block pointer
A = 0 => get date and time
1 => set date and time
IX = New time value (only if A=1)
HL = New date value (only if A=1)

Results:
A = Error
DE = Current file time value
HL = Current file date value

If A=1 then this function sets the date and time of last modification of the file or sub-directory specified by the drive/path/file string or fileinfo block. Global filename characters are not allowed in any part of the string so only one file can have its date and time modified by this function. If a device name such as "CON" is specified then no error will be returned but the device's date and time will not actually be changed.

The date and time format are exactly as contained in the directory entry and fileinfo blocks (see the "Program Interface Specification"). No checks are done for sensible dates or times, the values are simply stored. Note that if a fileinfo block is passed then the date and time stored in it will not be updated by this function.

If A=0 then the current values are just returned. Note that although the time value is passed in IX, it is returned in DE. The date and time of a file cannot be altered (although it can be read) if there is a file handle open to the file (".FOPEN" error).

3.62 DELETE FILE HANDLE

```
Parameters:   C = 52h (_HDELETE)
              B = File handle
Results:      A = Error
```

This function deletes the file associated with the specified file and closes the file handle. A file handle cannot be deleted if there are any other separately opened file handles open to the same file (".FOPEN" error). If there are any duplicates of the file handle (created by a "duplicate file handle" or "fork" function), then these duplicates will be marked as invalid and any attempt to use them will produce an ".HDEAD" error.

The error conditions for this function are the same as for the "delete file or sub-directory" function (function 4Dh). The file handle will always be closed, even if there is an error condition such as ".FILRO" or ".FOPEN".

3.63 RENAME FILE HANDLE

```
Parameters:   C = 53h (_HRENAME)
              B = File handle
             HL = New filename ASCIIZ string
Results:      A = Error
```

This function renames the file associated with the specified file handle with the new name in the string pointed to by HL. Apart from the fact that the file is specified by a file handle rather than an ASCIIZ string or a fileinfo block, this function is identical to the "rename file or subdirectory" function (function 4Eh), and has the same error conditions.

A file handle cannot be renamed if there are any other separately opened file handles for this file (".FOPEN" error), although it can be renamed if there are copies of this file handle, and in this case the copies will be renamed. Renaming a file handle will not alter the file pointer but it will do an implicit "ensure" operation.

3.64 MOVE FILE HANDLE

```
Parameters:   C = 54h (_HMOVE)
              B = File handle
             HL = New path ASCIIZ string
Results:      A = Error
```

This function moves the file associated with the specified file handle to the directory specified by the new path string pointed to by HL. Apart from the fact that the file is specified by a file handle rather than an ASCIIZ string or a fileinfo block, this function is identical to the "move file or subdirectory" function (function 4Fh), and has the same error conditions.

A file handle cannot be moved if there are any other separately opened file handles for this file (".FOPEN" error), although it can be moved if there are copies of this file handle, and in this case the copies will also be moved. Moving a file handle will not alter the file pointer but it will do an implicit "ensure" operation.

3.65 GET/SET FILE HANDLE ATTRIBUTES

Parameters:
C = 55h (_HATTR)
B = File handle
A = 0 => get attributes
1 => set attributes
L = New attributes byte (only if A=1)

Results:
A = Error
L = Current attributes byte

This function gets or sets the attributes byte of the file associated with the specified file handle. Apart from the fact that the file is specified by a file handle rather than an ASCIIZ string or a fileinfo block, this function is identical to the "get/set file attributes" function (function 50h), and has the same error conditions.

A file handle cannot have its attributes changed (although they can be read) if there are any other separately opened file handles for this file (".FOPEN" error). The file pointer will not be altered but an implicit "ensure" operation will be done.

3.66 GET/SET FILE HANDLE DATE AND TIME

Parameters:
C = 56h (_HFTIME)
B = File handle
A = 0 => get date and time
1 => set date and time
IX = New time value (only if A=1)
HL = New date value (only if A=1)

Results:
A = Error
DE = Current file time value
HL = Current file date value

This function gets or sets the date and time of the file associated with the specified file handle. Apart from the fact that the file is specified by a file handle rather than an ASCIIZ string or a fileinfo block, this function is identical to the "get/set file date and time" function (function 51h), and has the same error conditions.

A file handle cannot have its date and time changed (although they can be read) if there are any other separately opened file handles for this file (".FOPEN" error). The file pointer will not be altered but an implicit "ensure" operation will be done.

3.67 GET DISK TRANSFER ADDRESS

```
Parameters:     C = 57h (_GETDTA)
Results:       DE = Current disk transfer address
```

This function returns the current disk transfer address. This address is only used for the "traditional" CP/M style FCB functions and the absolute sector read and write functions.

3.68 GET VERIFY FLAG SETTING

```
Parameters:     C = 58h (_GETVFY)
Results:        B = 00h => verify disabled
                    FFh => verify enabled
```

This function simply returns the current state of the verify flag which can be set with MSX-DOS function 2Eh.

3.69 GET CURRENT DIRECTORY

```
Parameters:     C = 59h (_GETCD)
                B = Drive number (0=default, 1=A: etc)
               DE = Pointer to 64 byte buffer
Results:        A = Error
               DE = Filled in with current path
```

This function simply gets an ASCIIZ string representing the current directory of the specified drive into the buffer pointed to by DE. The string will not include a drive name or a leading or trailing "\" character, so the root directory is represented by a null string. The drive will be accessed to make sure that the current directory actually exists on the current disk, and if not then the current directory will be set back to the root and a null string returned.

3.70 CHANGE CURRENT DIRECTORY

```
Parameters:     C = 5Ah (_CHDIR)
               DE = Drive/path/file ASCIIZ string
Results:        A = Error
```

The drive/path/file string must specify a directory rather than a file. The current directory of the relevant drive will be changed to be this directory. If the specified directory does not exist then the current setting will be unaltered and a ".NODIR" error returned.

3.71 PARSE PATHNAME

Parameters:	C	= 5Bh (_PARSE)
	B	= Volume name flag (bit 4)
	DE	= ASCIIZ string for parsing
Results:	A	= Error
	DE	= Pointer to termination character
	HL	= Pointer to start of last item
	B	= Parse flags
	C	= Logical drive number (1=A: etc)

This function is purely a string manipulation function, it will not access the disks at all and it will not modify the user's string at all. It is intended to help transient programs in parsing command lines.

The volume name flag (which is in the same bit position as the volume name bit in an attributes byte) determines whether the string will be parsed as a "drive/path/file" string or a "drive/volume" string.

The pointer returned in DE will point to the first character which is not valid in a pathname string, and may be the null at the end of the string. See the "Command Specification" for details of the syntax of pathname strings and also for a list of valid characters.

The pointer returned in HL points to the first character of the last item (the "filename" component) of the string. For example if the string passed is "A:\XYZ\P.Q /F", then DE will point to the space character, and HL will point to the "P". If the parsed string ends with a "\" character, or is null (apart from a drive name) then HL will point to the same character as DE since in these cases there is no "last item". This special case will usually require some special handling in any program which uses this function.

The drive number returned in register C is the logical drive specified in the string. If the string did not start with a drive letter then register C will contain the default drive number, since the default drive has been implicitly specified. Register C will never be zero.

The parse flags returned in register B indicate various useful things about the string. For a volume name bits 1, 4, 5, 6 and 7 will always be clear. For a filename, bits 3 to 7 relate to the last item on the string (the "filename" component). The bit assignments are as follows:

b0 - set if any characters parsed other than drive name
b1 - set if any directory path specified
b2 - set if drive name specified
b3 - set if main filename specified in last item
b4 - set if filename extension specified in last item
b5 - set if last item is ambiguous
b6 - set if last item is "." or ".."
b7 - set if last item is ".."

3.72 PARSE FILENAME

Parameters:	C = 5Ch (_PFILE)
	DE = ASCIIZ string for parsing
	HL = Pointer to 11 byte buffer
Results:	A = Error (always zero)
	DE = Pointer to termination character
	HL = Preserved, buffer filled in
	B = Parse flags

This function is purely a string manipulation function, it will not access disks at all and will not modify the string at all. It is intended mainly to help transient programs in printing out filenames in a formatted way. The ASCIIZ string will be parsed as a single filename item, and the filename will be stored in the user's 11 byte buffer in expanded form, with both the filename and the extension padded out with spaces.

The parse flags returned in register B are identical to those for the "parse pathname" function above (function 5Bh), except that bits 0, 1 and 2 will always be clear. The user's buffer will always be filled in, even if there is no valid filename in the string, in which case the buffer will be filled with spaces. "*" characters will be expanded to the appropriate number of "?"s. If either the filename or the filename extension is too long then the excess characters will be ignored.

The pointer returned in register DE will point to the first character in the string which was not part of the filename, which may be the null at the end of the string. This character will never be a valid filename character (see the "Command Specification" for details of valid filename characters).

3.73 CHECK CHARACTER

Parameters:	C = 5Dh (_CHKCHR)
	D = Character flags
	E = Character to be checked
Results:	A = 0 (never returns an error)
	D = Updated character flags
	E = Checked (upper cased) character

This function allow language independent upper casing of characters and also helps with handling 16-bit characters and manipulation of filenames. The bit assignments in the character flags are as follows:

```
   b0    - set to suppress upper casing
   b1    - set if first byte of 16-bit character
   b2    - set if second byte of 16-bit character
   b3    - set => volume name (rather than filename)
   b4    - set => not a valid file/volume name character
b5...b7 - reserved (always clear)
```

Bit 0 is used to control upper casing. If it is set then the character will be upper cased according to the language setting of the machine. If this bit is clear then the returned character will always be the same as the character passed.

The two 16-bit character flags (bits 1 and 2) can both be clear when the first character of a string is checked and the settings returned can be passed straight back to this function for each subsequent character. Care must be taken with these flags when moving backwards through strings which may contain 16-bit characters.

Bit 4 is set on return if the character is one of the set of filename or volume name terminator characters. Bit 3 is simply used to determine whether to test for filename or volume name characters since the sets are different. 16-bit characters (either byte) are never considered as volume or filename terminators.

3.74 GET WHOLE PATH STRING

Parameters:	C = 5Eh (_WPATH)
	DE = Pointer to 64 byte buffer
Results:	A = Error
	DE = Filled in with whole path string
	HL = Pointer to start of last item

This function simply copies an ASCIIZ path string from an internal buffer into the user's buffer. The string represents the whole path and filename, from the root directory, of a file or sub-directory located by a previous "find first" or "find new" function. The returned string will not include a drive, or an initial "\" character. Register HL will point at the first character of the last item on the string, exactly as for the "parse path" function (function 5Bh).

If a "find first" or "find new" function call is done with DE pointing to an ASCIIZ string then a subsequent "get whole path" function call will return a string representing the sub-directory or file corresponding to the fileinfo block returned by the "find" function. If this is a sub-directory then the fileinfo block may be passed back in register DE to another "find first" function call, which will locate a file within this sub-directory. In this case the newly located file will be added onto the already existing whole path string internally, and so a subsequent "get whole path string" function call will return a correct whole path string for the located file.

Great care must be taken in using this function because the internal whole path string is modified by many of the function calls, and in many cases can be invalid. The "get whole path" function call should be done immediately after the "find first" or "find new" function to which it relates.

3.75 FLUSH DISK BUFFERS

Parameters:	C = 5Fh (_FLUSH)
	B = Drive number (0=default, FFh=all)
	D = 00h => Flush only
	= FFh => Flush and invalidate
Results:	A = Error

This function flushes any dirty disk buffers for the specified drive, or for all drives if B=FFh. If register D is FFh then all buffers for that drive will also be invalidated.

3.76 FORK TO CHILD PROCESS

Parameters:	C = 60h (_FORK)
Results:	A = Error
	B = Process id of parent process

This function informs the system that a child process is about to be started. Typically this is a new program or sub-command being executed. For example COMMAND2.COM does a "fork" function call before executing any command or transient program.

A new set of file handles is created, and any current file handles which were opened with the "inheritable" access mode bit set (see the "open file handle" function - function 43h), are copied into the new set of file handles. Any file handles which were opened with the "inheritable" bit clear will not be copied and so will not be available to the child process. The standard file handles (00h...05h) are inheritable and so these will be copied.

A new process id is allocated for the child process and the process id. of the parent process is returned so that a later "join" function call can switch back to the parent process. A ".NORAM" error can be produced by this function if there is insufficient memory to duplicate the file handles.

Because the child process now has a copy of the previous file handles rather than the originals, if one of them is closed then the original will remain open. So for example if the child process closes the standard output file handle (file handle number 1) an re-opens it to a new file, then when a "join" function is done to return to the parent process the original standard output channel will still be there.

3.77 REJOIN PARENT PROCESS

Parameters:	C = 61h (_JOIN)
	B = Process id of parent, or zero
Results:	A = Error
	B = Primary error code from child
	C = Secondary error code from child

This function switches back to the specified parent process and returns the error code which the child process terminated with in register B, and a secondary error code from the child in register C. Although the relationship between parent and child processes is strictly one-to-one, this function can jump back several levels by giving it a suitable process id. A ".IPROC" error will be returned if the process id is invalid.

The child process's set of file handles are automatically closed and the parent process's set of file handles becomes active again. Also any user RAM segments which the child process had allocated will be freed.

If the process id passed to this function is zero then a partial system re-initialisatin is done. All file handles are closed and the standard input and output handles re-opened. All user segments are freed, all environment strings are cleared and the buffered line input circular buffer is cleared. This should not normally be done by a user program if it intends to return to the command interpreter since the command interpreter will not be in a consistent state after this.

This function takes great care that the freeing of memory and adjusting of process id is done before actually closing any file handles and thus before accessing the disk. This ensures that if a disk error occurs and is aborted, the join operation will have been done successfully. However if a "join 0" produces a disk error which is aborted, then the re-initialization of default file handles, environment and character buffer will not have been done. In this case another "join 0" function call should be done and this will not attempt access disk (because all the files have been closed) and so will be successful.

Note that if this function call is made via 0F37Dh then registers B and C will not return the error codes. This is because program termination and abort handling must be done by the application program. The error code will have been passed to the abort vector and code there must remember the error code if it needs to. See the "terminate with error code" function (function 62h) for the meaning of the primary and secondary error code.

3.78 TERMINATE WITH ERROR CODE

Parameters: C = 62h (_TERM)
B = Error code for termination
Results: Does not return

This function terminates the program with the specified error code, which may be zero indicating no error. This function call will never return to the caller (unless a user abort routine executes forces it to - see function 63h). The operation of this function is different depending on whether it was called from the MSX-DOS environment via 00005h or from the disk BASIC environment via 0F37Dh.

If called via 00005h then if a user abort routine has been defined by function 63h it will be called with the specified error code (and a zero secondary error code). Assuming that this routine returns, or if there was no user abort routine defined, then control will be passed back to whatever loaded the transient program via a jump at location 00000h. This will almost always be the command interpreter, but in some cases it may be another transient program. The error code will be remembered by the system and the next "join" function (function 61h) which is done will return this error code. The command interpreter will print an error message for any code in the range 20h...FFh, but will not print a message for errors below this.

If this function is called from the disk BASIC environment via 0F37Dh then control will be passed to the abort vector at location "ABVECT". In this environment there is no separately defined user abort routine and the error code must be remembered by the code at "ABVECT" because "join" will not return the error code.

3.79 DEFINE ABORT EXIT ROUTINE

Parameters:	C = 63h (_DEFAB)
	DE = Address of abort exit routine
	0000h to un-define routine
Results:	A = 0 (never generates errors)

This function is only available when called via location 00005h in the MSX-DOS environment. It cannot be called at location 0F37Dh from the disk BASIC environment.

If register DE is zero then a previously defined abort routine will be undefined, otherwise a new one will be defined. The abort routine will be called by the system whenever the transient program is about to terminate for any reason other than a direct jump to location 0000h. Programs written for MSX-DOS 2.00 should exit with a "terminate with error code" function call (function 061h) rather than a jump to location 0000h.

The user abort routine will be entered with the user stack active, with IX, IY and the alternate register set as it was when the function call was made and with the whole TPA paged in. The termination error code will be passed to the routine in register A with a secondary error code in register B and if the routine executes a "RET" then the values returned in registers A and B will be stored as the error codes to be returned by the "join" function, and normally printed out by the command interpreter. Alternatively the routine may jump to some warm start code in the transient program rather than returning. The system will be in a perfectly stable state able to accept any function calls.

The primary error code passed to the routine in register A will be the code which the program itself passed to the "terminate with error code" function (which may be zero) if this is the reason for the termination. The routine will also be called if a Ctrl-C or Ctrl-STOP is detected (".CTRLC" or ".STOP" error), if a disk error is aborted (".ABORT" error), or if an error occurred on one of the standard input or output channels being accessed through MSX-DOS function calls 01h...0Bh (".INERR" or ".OUTERR").

The errors ".ABORT", ".INERR" and ".OUTERR" are generated by the system as a result of some other error. For example a ".ABORT" can result from a ".NRDY" error, or a ".INERR" can result from a ".EOF" error. In these cases the original error code (".NRDY" or ".EOF" is passed to the abort routine in register B as the secondary error code. For all other errors there is no secondary error code and register B will be zero.

If the abort routine executes "POP HL : RET" (or equivalent) rather than a simple return, then control will pass to the instruction immediately following the MSX-DOS call or BIOS call in which the error occurred. This may be useful in conjunction with a disk error handler routine (see function 64h) to allow an option to abort the current MSX-DOS call when a disk error occurs.

3.80 DEFINE DISK ERROR HANDLER ROUTINE

Parameters: C = 64h (_DEFER)
DE = Address of disk error routine
0000h to un-define routine

Results: A = 0 (never generates errors)

This function specifies the address of a user routine which will be called if a disk error occurs. The routine will be entered with the full TPA paged in, but with the system stack in page-3 active and none of the registers will be preserved from when the MSX-DOS function call was made.

The error routine can make MSX-DOS calls but must be very careful to avoid recursion, especially for compatibility with future systems which may allow redirection of standard input and output. The list of function calls in section 2 of this document indicates which function calls can be safely made from a user error routine. Particular care must be taken in doing character I/O in case the standard I/O channels have been redirected. See the "get/set redirection state" function (function 70h) for details of this.

The specification of parameters and results for the routine itself is as below. All registers including IX, IY and the alternate register set may be destroyed but the paging and stack must be preserved. The routine must return to the system, it must not jump away to continue the transient program. If it wants to do this then it should return A=1 ("abort") and a user abort routine will then get control and this may do whatever it wants to.

```
Parameters:   A = Error code which caused error
              B = Physical drive
              C = b0 - set if writing
                  b1 - set if ignore not recommended
                  b2 - set if auto-abort suggested
                  b3 - set if sector number is valid
             DE = Sector number (if b3 of C is set)
Results:      A = 0 => Call system error routine
                  1 => Abort
                  2 => Retry
                  3 => Ignore
```

3.81 GET PREVIOUS ERROR CODE

```
Parameters:   C = 65h (_ERROR)
Results:      A = 0
              B = Error code from previous function
```

This function allows a user program to find out the error code which caused the previous MSX-DOS function call to fail. It is intended for use with the old CP/M compatible functions which do not return an error code. For example if a "create file FCB" function returns A=0FFh thee could be many reasons for the failure and doing this function call will return the appropriate on, for example ".DRFUL" or ".SYSX".

3.82 EXPLAIN ERROR CODE

Parameters:	C	= 66h (_EXPLAIN)
	B	= Error code to be explained
	DE	= Pointer to 64 byte string buffer
Results:	A	= 0
	B	= 0 or unchanged
	DE	= Filled in with error message

This function allows a user program to get an ASCIIZ explanation string for a particular error code returned by any of the MSX-DOS functions. If an error comes from one of the old functions then "get previous error code" must be called first to get the real error code and then this function can be called to get an explanation string.

The "Program Interface Specification" contains a list of all the currently defined error codes and the messages for them. Foreign language versions of the system will of course have different messages. If the error code does have a built in explanation string then this string will be returned and register B will be set to zero. If there is no explanation string then a string of the form: "System error 194" or "User error 45" will be returned, and register B will be unchanged. (System errors are those in the range 40h...FFh and user errors are 00h...3Fh.)

3.83 FORMAT A DISK

```
Parameters:   C = 67h (_FORMAT)
              B = Drive number (0=>default, 1=>A:)
              A =    00h     => return choice string
                   01h...09h => format this choice
                   0Ah...FEh => illegal
                      FFh    => new boot sector
             HL = Pointer to buffer (if A=1...9)
             DE = Size of buffer (if A=1...9)
Results:      A = Error
              B = Slot    \   of choice string
             HL = Address / (only if A=0 on entry)
```

This function is used to format disks and is really only provided for the "FORMAT" command although other programs may use it (with care) if they find it useful. It has three different options which are selected by the code passed in register A.

If A=0 then registers B and HL return the slot number and address respectively of an ASCIIZ string which specifies the choice of formats which is available. A ".IFORM" error will be returned if this disk cannot be formatted (for example the RAM disk). Normally the string will be read using the "RDSLT" routine and displayed on the screen followed by a "? " prompt. The user then specifies a choice "1"..."9" and this choice is passed back to the "format" function, after a suitable warning prompt, to actually format the disk. There is no way of knowing what disk format a particular choice refers to since this is dependant on the particular disk driver and there is no standardization.

If A=01h...09h then this is interpreted as a format choice and a disk will be formatted in the specified drive with no further prompting. Register HL and DE must specify a buffer area to be used by the disk driver. There is no way of knowing how big this buffer should be so it is best to make it as big as possible. If the buffer crosses page boundaries then this function will select the largest portion of it which is in one page for passing to the disk driver. Many disk drivers do not use this buffer at all.

If A=FFh then the disk will not actually be formatted, but it will be given a new boot sector to make the disk a true MSX-DOS 2.00 disk. This is designed to update old MSX-DOS 1.0 disks to have a volume id and thus allow the full disk checking and undeletion which MSX-DOS 2.00 allows. Also there are some MSX-DOS 1.0 implementations which put an incorrect boot sector on the disk and these disks cannot be used by MSX-DOS 2.00 until they have been corrected by this function.

The "new boot sector" function is mainly intended for the "FIXDISK" utility program, but may be used by other programs if they find it useful. If it is used then a "get format choice" function call (A=0) should be done first and if this returns an error (typically ".IFORM") then the operation should be aborted because this is a drive which does not like to be formatted and the disk could be damaged by this function.

3.84 CREATE OR DESTROY RAMDISK

Parameters:	C = 68h (_RAMD) B = 00h => destroy RAM disk 1...FEh => create new RAM disk FFh => return RAM disk size
Results:	A = Error B = RAM disk size

If register B=0FFh then this routine just returns the number of 16k RAM segments which are allocated to the RAM disk currently. A value of zero indicates that there is no RAM disk currently defined. If B=0 then the current RAM disk will be destroyed, loosing all data which it contained and no error will be returned if there was no RAM disk.

Otherwise, if B is in the range 01h...FEh then this function will attempt to create a new RAM disk using the number of 16k segments specified in register B. An error will be returned if there is already a RAM disk (".RAMDX") or if there is not even one segment free (".NORAM"). If there are insufficient free RAM segments to make a RAM disk of the specified size then the largest one possible will be created. No error is returned in this case.

In all cases the size of the RAM disk will be returned in register B as a number of segments. Note that some of the RAM is used for the file allocation tables and the root directory so the size of the RAM disk as indicated by "DIR" or "CHKDSK" will be somewhat smaller than the total amount of RAM used. The RAM will always be assigned the drive letter "H:" regardless of the number of drives in the system.

3.85 ALLOCATE SECTOR BUFFERS

Parameters:	C = 69h (_BUFFER) B = 0 => return number of buffers else number of buffers required
Results:	A = Error B = Current number of buffers

If B=0 then this function just returns the number of sector buffers which are currently allocated. If B<>0 then this function will attempt to use this number of sector buffers (must always be at least 5). If it cannot allocate as many as requested then it will allocate as many as possible and return the number in register B but will not return an error. The number of sector buffers can be reduced as well as increased.

The sector buffers are allocated in a 16k RAM segment outside the normal 64k so the number of buffers does not detract from the size of the TPA. However the number of buffers does affect efficiency since with more buffers allow more FAT and directory sectors to be kept resident. The maximum number of buffers will be about 20.

3.86 LOGICAL DRIVE ASSIGNMENT

Parameters:	C = 6Ah (_ASSIGN)
	B = Logical drive number (1=A: etc)
	D = Physical drive number (1=A: etc)
Results:	A = Error
	D = Physical drive number (1=A: etc)

This function controls the logical to physical drive assignment facility. It is primarily intended for the "ASSIGN" command although user programs may want to use it to translate logical drive numbers to physical drive numbers.

If both B and D are non-zero then a new assignment will be set up. If register B is non-zero and register D is zero then any assignment for the logical drive specified by B will be cancelled. If both register B and D are zero then all assignments will be cancelled. If register B is non-zero and register D is FFh then the current assignment for the logical drive specified by register B will simply be returned in register D.

All drives used in the various function calls, including drive names in strings and drive numbers as parameters to function calls, are logical drives. However the drive number passed to disk error routines is a physical drive so if "ASSIGN" has been used these may be different from the corresponding logical drive.

3.87 GET ENVIRONMENT ITEM

Parameters:	C = 6Bh (_GENV)
	HL = ASCIIZ name string
	DE = Pointer to buffer for value
	B = Size of buffer
Results:	A = Error
	DE = Preserved, buffer filled in if A=0

This function gets the current value of the environment item whose name is passed in register HL. A ".IENV" error is returned if the name string is invalid. If there is no environment item of that name then a null string will be returned in the buffer. If there is an item of that name then its value string will be copied to the buffer. If the buffer is too small then the value string will be truncated with no terminating null and a ".ELONG" error will be returned. A buffer 255 bytes will always be large enough since value strings cannot be longer than this (including the terminating null).

3.88 SET ENVIRONMENT ITEM

Parameters:	C = 6Ch (_SENV)
	HL = ASCIIZ name string
	DE = ASCIIZ value string
Results:	A = Error

This function sets a new environment item. If the name string is invalid then a ".IENV" error is returned, otherwise the value string is checked and a ".ELONG" error returned if it is longer than 255 characters, or a ".NORAM" error if there is insufficient memory to store the new item. If all is well then any old item of this name is deleted and the new item is added to the beginning of the environment list. If the value string is null then the environment item will be removed.

3.89 FIND ENVIRONMENT ITEM

Parameters:	C = 6Dh (_FENV)
	DE = Environment item number
	HL = Pointer to buffer for name string
Results:	A = Error
	HL = Preserved, buffer filled in

This function is used to find out what environment items are currently set. The item number in register DE identifies which item in the list is to be found (the first item corresponds to DE=1). If there is an item number <DE> then the name string of this item will be copied into the buffer pointed to by HL. If the buffer is too small then the name will be truncated with no terminating null, and a ".ELONG" error returned. A 255 byte buffer will never be too small. If there is no item number <DE> then a null string will be returned, since an item can never have a null name string.

3.90 GET/SET DISK CHECK STATUS

```
Parameters:   C = 6Eh (_DSKCHK)
              A = 00h => get disk check status
                  01h => set disk check status
              B = 00h => enable  \
                  FFh => disable / only if A=01h
Results:      A = Error
              B = Current disk check setting
```

If A=0 then the current value of the disk check variable is returned in register B. If A=01h then the variable is set to the value in register B. A value of 00h means that disk checking is enabled and a non-zero means that it is disabled. The default state is enabled.

The disk check variable controls whether the system will re-check the boot sector of a disk to see whether it has changed, each time a file handle, fileinfo block or FCB is accessed. If it is enabled then it will be impossible to accidentally access the wrong disk by changing a disk in the middle of an operation, otherwise this will be possible and may result in a corrupted disk. Depending on the type of disk interface, there may be some additional overhead in having this feature enabled although with many types of disk (those with explicit disk change detection hardware) it will make no difference and the additional security is well worth having.

3.91 GET MSX-DOS VERSION NUMBER

```
Parameters:   C = 6Fh (_DOSVER)
Results:      A = Error (always zero)
             BC = MSX-DOS kernel version
             DE = MSXDOS2.SYS version number
```

This function allows a program to determine which version of MSX-DOS it is running under, in case it uses version specific features. Two version numbers are returned, one in BC for the MSX-DOS kernel in ROM and the other is DE for the MSXDOS2.SYS system file. Both of these version numbers are BCD values with the major version number in the high byte and the two digit version number in the low byte. For example if there were a version 2.34 of the system (which there is not), it would be represented as 0234h.

For compatibility with MSX-DOS 1.0, the following procedure should always be followed in using this function. Firstly if there is any error (A<>0) then it is not MSX-DOS at all. Next look at register B. If this is less than 2 then the system is earlier than 2.00 and registers C and DE are undefined. If register B is 2 or greater then registers BC and DE can be used as described above. In general the version number which should be checked (after this procedure) is the MSXDOS2.SYS version in register DE.

3.92 GET/SET REDIRECTION STATE

Parameters:
C = 6Eh (_DSKCHK)
A = 00h => get redirection state
01h => set redirection state
B = New state. b0 - standard input
b1 - standard output

Results:
A = Error
B = Redirection state before command
b0 set => input is redirected
b1 set => output is redirected

This function is provided primarily for disk error routines and other character I/O which must always go to the console regardless of any redirection. When the CP/M character functions (functions 01h...0Bh) are used, they normally refer to the console. However if the standard input or output file handles (file handles 0 and 1) have been closed and reopened to a disk file, then the CP/M character functions will also go to the disk file. However certain output such as disk error output must always go to the screen regardless.

This function allows any such redirection to be temporarily cancelled by calling this function with A=1 and B=0. This will ensure that any subsequent CP/M console I/O will go to the console, and will also return the previous setting so that this can be restored afterwards. The system is is a somewhat unstable state when the redirection state has been altered like this and there are many function calls which will reset the redirection to its real state over-riding this function. In general any function call which manipulates file handles, such as "open", "close", "duplicate" and so on, will reset the redirection state. The effect of this function is therefore purely temporary.

++++++++++ END OF DOCUMENT ++++++++++